N90-29819

Learning in Tele-autonomous Systems using Soar*

John E. Laird Eric S. Yager Christopher M. Tuck
Michael Hucka
Artificial Intelligence Laboratory
The University of Michigan
Ann Arbor, MI 48109-2110

Abstract

Robo-Soar is a high-level robot arm control system implemented in Soar. Robo-Soar learns to perform simple block manipulation tasks using advice from a human. Following learning, the system is able to perform similar tasks without external guidance. It can also learn to correct its knowledge, using its own problem solving in addition to outside guidance. Robo-Soar corrects its knowledge by accepting advice about relevance of features in its domain, using a unique integration of analytic and empirical learning techniques.

1 Introduction

Tele-robotics allows for intelligent action at a distance. A human, with a long history of solving manipulation and construction tasks, can control a remotely located robot without incurring the risks or costs associated with the environment. However, if the task is repetitive and boring, or if there is sufficient time-delay between the human's instructions and the robot's actions, the robot will have to assume more of the responsibility for intelligent action [5]. By endowing the robot with some intelligence, it should be able to handle routine operations, as well as respond quickly to emergency situations. But how does the robot become intelligent?

Learning through interacting with a human is one way to increase the knowledge of a robot. Initially, a robot may have only very general abilities and must be tightly controlled by a human operator. Through its experiences, the robot can become more and more autonomous. It can increase its repertoire of methods for solving problems, improve its reaction time to events in the environment, and learn to notice new properties of objects in the environment. When a problem is sufficiently novel, so that the robot is unable to easily solve it, the robot can request guidance from a human. The robot can then learn from the interaction so that future intervention is not necessary. If human advice is not readily available, the robot can attempt to solve the problem itself, searching through the space of possible safe actions until it finds a solution. During this problem solving it can learn to avoid actions that do not lead to a solution as well as learn appropriate actions for solving a given class of problems.

Although learning is important in creating intelligent autonomous systems, it can not be considered in isolation, to be grafted onto an existing architecture at a later time The architecture must be able to attack problems when it has very little knowledge and requires significant outside guidance, as well as when it has large amounts of knowledge and no outside guidance is required. The architecture we use is Soar, an architecture with proven general problem solving and learning capabilities [14]. Soar's learning mechanism, called chunking, is completely integrated with its problem solving so that it learns during problem solving [15].

In this paper we describe initial progress in learning by a tele-autonomous system called Robo-Soar. Robo-Soar performs simple block manipulation task using a Puma robot arm and a machine vision system. We demonstrate the acquisition of new control knowledge so that the system is able to directly solve problems that previously required combinatorial search. We also demonstrate that Robo-Soar is able to learn to correctly manipulate new objects which are not covered by its original knowledge base. This is made possible by

*This research was sponsored by grant NCC2-517 from NASA Ames and ONR grant N00014-88-K-0554.

extending the interface between the human and robot so that the human can point out important features of the environment in addition to providing motor commands.

2 Related Work

Many approaches are possible for learning from experience and outside guidance in tele-autonomous systems. In the simplest case, the human is restricted to controlling the robot's actions, and the learning system must watch "over the shoulder" of the human as the problem is solved. This is the scheme used in robotic programming systems where a human leads the system through a fixed set of commands to achieve a goal. When only the exact commands are stored, the system can only perform that one task because no association is created between the goal and the actions being performed. Another disadvantage is that there is no conditionality in the learned plan. The robot will do exactly the learned set of actions, independent of the state of the environment.

To avoid these problems, "learning apprentices" have been developed that create generalized plans, indexed by the appropriate goal. These systems, such as LEAP [17], are based on a learning strategy called explanation-based learning (EBL) [6,16]. EBL has its roots in the macro-operator learning mechanism of STRIPS [8]. In these systems, an underlying "domain theory" is used to "explain" the actions of the human expert. From the derived explanation, all of the dependencies between the actions are recovered and a general plan is created.

The ARMS system developed by Alberto Segre uses EBL to learn generalized plans for simulated manipulator control [24,23]. The input to the system was the sequence of manipulator moves necessary to construct a specific example of a simple object, such as a revolute joint. Through analysis of the sequence and its own underlying theory of the domain, the system was able to learn general plans that would be independent of the specific example. Boy and Delail [2] have used a similar approach for training a system that advises a human tele-robotic controller.

Our goal is to extend the ARMS approach in the following ways:

1. Solve a problem involving interaction with a real environment. In a real environment, perception is incomplete, so that movements may obscure the current view. In addition, actions and perception are not instantaneous; the system must sometimes wait.

2. Modify the interaction so that the system has more control of the interaction. Therefore, the human will no longer provide a monolithic plan for solving the problem. Instead, the human will give guidance only when the system needs it, or when the human wants. The advantage of this approach is that it minimizes human interaction, and also provides a context for the human's response.

3. Learn individual rules for each decision of the plan instead of learning a complete plan. When plans are learned, it is difficult for the performance system that uses the plans to react quickly to changes in the environment that invalidate the plan. Our goal is to have a system that uses the knowledge from all of its prior experiences for each decision it makes so that, effectively, it dynamically combines portions of independently learned plans.

4. Integrate learning through guidance with general problem solving and autonomous learning. The system should be able to learn from its own experiences, avoiding interaction with humans for simple problems it can solve on its own.

5. Extend the knowledge about the domain. In general, EBL systems are completely dependent on the knowledge encoded in their domain theory. Given the complexity of the world, any finite domain theory for a significant component of the real world will be incomplete and incorrect. We will demonstrate that it is possible to extend an incomplete theory of the domain using a combination of analytic and empirical learning techniques.

Previous work in Soar has already attacked extensions 2, 3 and 4 [9]. In this paper, we demonstrate that the same approach can be extended to real environments where the system's initial knowledge may be incomplete or overgeneral.

We will investigate simple block manipulation tasks. The goal of the robot is to line-up a set of small blocks that have been scattered over the work area. For the first task, all of the blocks are simple cubes that

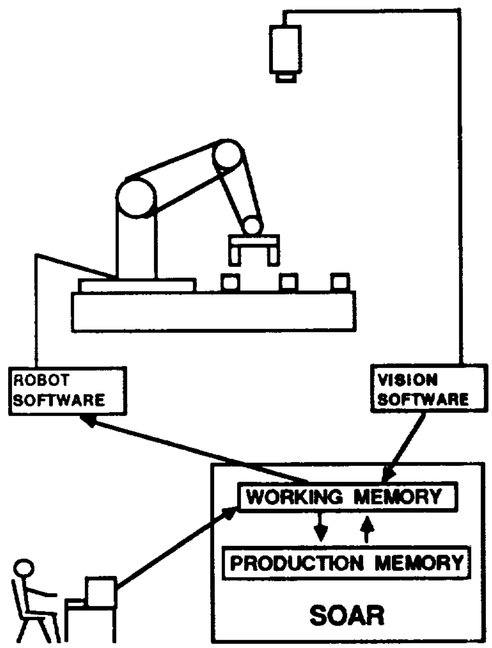

Figure 1: Robo-Soar system architecture.

the gripper can pick up in two different orientations as shown in Figure 1. In the second task, there is a block that is a triangular prism. The gripper is unable to pick up the prism when it closes over the inclined sides. Instead it must be oriented so that it closes over the vertical faces of the block.

3 System Architecture

Figure 1 shows the basic Soar architecture on which Robo-Soar is built.[1] Input comes from a camera mounted directly above the work area. A separate computer processes the images, providing asynchronous input to the rest of the system. The vision processing extracts the positions and orientations of the blocks in the work area as well as distinctive features of the blocks, such as its number. Soar accepts new visual and gripper information whenever it arrives.

In Soar, a task is solved by searching through a problem space of possible states, applying operators to transform an initial state to some desired state. For the block manipulation task, the states are different configurations of the blocks and gripper in the external environment. Some basic operators are shown in the trace of Robo-Soar solving a simple block manipulation problem in Figure 2. These operators correspond directly to commands sent to the robot controller. Robo-Soar solves a problem by selecting operators until it achieves the goal. When an operator is selected, motor commands are sent to the Puma controller and executed. One complication is that the camera is mounted directly above the work area so that the arm obscures the view of a block that is being picked up. Two operators, snap-in and snap-out, move the arm in and out of the work area so that a clear image can be obtained. These operators are necessary, but for simplicity, they will not be included in any of the examples.

This characterization of Robo-Soar does not distinguish it from any other robot controller. What is different is the way Soar makes the decisions to select an operator. Many AI or robotic systems create a plan of actions that the robot must execute. Instead of creating a plan, Soar makes each decision based on a consideration of its long term knowledge, its perception of the environment, and its own internal goals. In this way, it is similar to "situated action" systems [25] such as Pengi [1], and Brooks' Creatures [3]. It represents

[1] Robo-Soar is implemented in Soar 5.0, a new version that allows internal operators to directly modify states [13]. In earlier versions of Soar, operators could only create new states, copying over those aspects of the prior state that did not change.

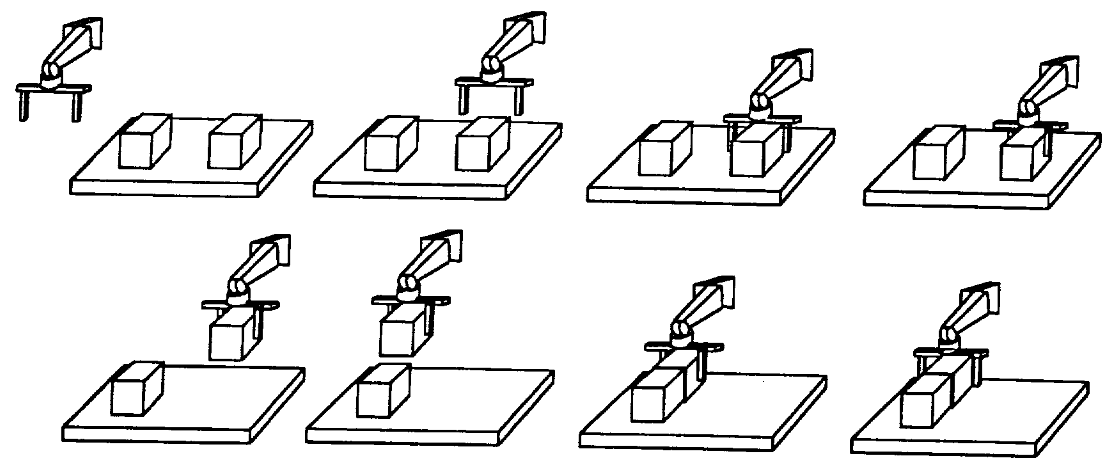

Figure 2: Moving a block using the primitive operators for block manipulation task.

the current situation and its current goals in *working memory*. Soar's long-term knowledge is represented as productions (condition-action rules) that are continually matched against working memory.[2] All of Soar's input and output pass through working memory so that productions can detect changes in the environment, or create expectations which relate to the effects of its actions [27].

In contrast to traditional production systems such as OPS5, Soar fires all successfully matched productions in parallel, allowing them to elaborate the current situation, or create *preferences* for the next action to be taken. There is a fixed preference language that allows productions to assert that operators are acceptable, not acceptable, better than other operators, as good as others and so on. Production firing continues until quiescence is reached (no additional productions match) so that short chains of monotonic inference are possible. Some productions act as bottom-up recognizers, quickly parsing incoming data and building up symbolic descriptions. Other productions compare these descriptions to the goals of the system and create preferences for the alternative operators. Following quiescence, Soar examines the preferences and selects the best operator for the given situation (possibly maintaining the current operator if its actions have not yet completed).

In a familiar domain, Soar's knowledge is adequate to pick and apply an appropriate operator. However, when Soar's preferences do not determine a best choice, or when it is unable to implement the selected operator directly, an *impasse* arises and Soar automatically generates a subgoal. In the subgoal, Soar uses the same approach; it casts the problem of resolving the impasse as a search through a problem space and uses its production memory to control the search when possible. The operators in the subgoal can modify or query the environment, or they may be completely internal, possibly simulating external operators on internal representations. Soar's production memory provides knowledge for selecting operators during the search as well as for implementing the actions of the internal operators. Impasses can arise while selecting or implementing internal operators, so that a hierarchy of subgoals is dynamically created. Within these subgoals, Soar can request advice from a human to help it determine the solution; if advice is not available, it will search.

When Soar creates results in its subgoals, it learns productions, called *chunks*, that summarize the processing that occurred. The actions of a chunk are based on the results of the subgoal. The conditions are based on those working-memory elements that were tested in the subgoal in order to derive the results. This technique is similar to explanation-based learning [6,16]. Although the technique is similar to EBL, most EBL systems learn a plan or schema of the actions in the subgoal. The plan is used to control problem solving in similar goals in the future. Soar's approach is different. The production learned for a goal does not provide control information for future problem solving in that goal. Instead, the chunk encapsulates the processing in the subgoal, eliminating the impasse that gave rise to the subgoal. Control knowledge is learned from subgoals created to select between competing operators.

[2] By exploiting parallelism, advanced production system implementations are very efficient at matching productions [10,11,26].

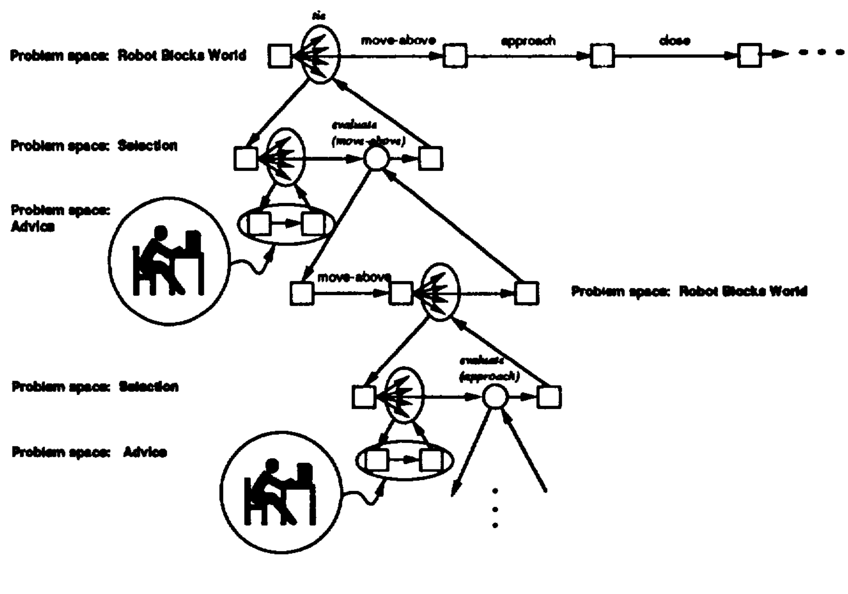

Figure 3: Trace of problem solving using external advice.

4 Learning Control Knowledge from External Advice

To apply Soar to a task, some initial knowledge about the operators in the task must be encoded in Soar. For the block manipulation tasks, this includes productions that suggest operators whenever they can legally be applied, as well as productions that can simulate the actions of the operators internally. In addition there is knowledge about the expected position of blocks following an action. Expectations have to be created because the vision system can not provide positive evidence that an operator applied correctly until it withdraws the arm and snaps it out of the way.

With just this basic knowledge, and no additional control knowledge, Robo-Soar can attempt a task, but it will encounter impasses whenever it tries to select a task operator, as shown in Figure 3. In this figure, horizontal arcs represent operator applications, while squares represent states. Downward pointing arcs are used to represent the creation of subgoals, and upward pointing arcs represent the termination of subgoals. In response to these *tie* impasses, Soar uses the *selection* problem space. Within the selection space, operators are created to evaluate the alternative operators. If the correct operator is evaluated, the system can pursue an internal search to ensure that it is on the path to the goal. Therefore, the efficiency of the search is determined by these decisions. The *advice* is used to obtain guidance from a human. In the advice problem space, the system waits for a human to provide advice. If advice is available, the system uses it to evaluate the appropriate operator. If no advice arrives while the system is waiting, it selects randomly.

Once an operator in the selection problem space has been selected to evaluate a task operator, another subgoal arises if no evaluation is directly available. In response to this impasse, Soar simulates the task operator on an internal copy of the external environment. If the resulting internal state achieves the goal, the operator is evaluated as being "best". If no evaluation is produced, the search continues to determine if another operator can lead Soar to the goal. This results in another impasse, and as should be evident, a depth-first search is performed recursively.

Once the goal is achieved within the internal search, preferences are created to select the operators being evaluated.[3] Each of these preferences is a result, and productions are built to summarize the processing that led to their creation. Figure 4 is an example of the production that is learned for the **approach** operator. Notice that it not only tests aspects of the current situation, but also aspects of the goal. The productions learned from this search are quite general and do not include any tests of the exact positions or names of the blocks. These features are not included because they were not tested in the subgoal. An important observation about chunking is that the generality of the learning is closely tied to the generality of the goals

[3] If the search leads to illegal or unacceptable states, a preference is created to avoid the responsible operator.

```
If the approach operator is applicable, and
   the gripper is holding nothing, in the safe plane above a block,
   and that block must be moved to achieve the goal,
then create a best preference for the approach operator.
```

Figure 4: Example production learned by Robo-Soar.

and the knowledge used to achieve the goals.

Following the look-ahead search and the accompanied learning, the system is able to directly solve the problem and similar problems with different initial block configurations. It has learned the appropriate operator to apply at each decision point in the search. However, this is not a blind application of a plan. Each of the productions that was learned will test aspects of the environment to insure that they are used only when appropriate. One result of this is that Robo-Soar automatically maintains an operator until sensory information gives it feedback that the situation has changed. In addition, if an operator has an unexpected result, Robo-Soar will not continue with the learned sequence of operators.

5 Refining Incorrect Knowledge

A problem with the traditional learning apprentice approach is that the learning is only as good as the underlying knowledge [20]. If there is an error in the original domain theory, the human has no avenue available for communicating corrections. This is a general problem with deductive learning techniques such as EBL. Although one could argue that errors in the original knowledge can be avoided through careful coding, the same problem can arise when an existing system encounters a problem outside its original specification.

We consider a simple case of this problem by attempting the same task as before except with blocks shaped as triangular prisms. If the original operators were implemented with only cubes in mind, all of the control knowledge and underlying simulation would not be sensitive to the fact that there is another feature in the input that must be attended to. To the Robo-Soar vision system, the prisms look just like cubes, except for a line down the middle at the apex of the triangle. In order to pick up these blocks, the gripper must be aligned with the vertical faces of the block, not just any two sides. If it is not correctly aligned, the gripper will close, but upon withdrawing the gripper, the block will not be picked up.

There are many possible approaches to this problem. First, the system could have an underlying theory of inclined planes, grippers, friction, etc. that it uses to understand why the block was not picked up. One problem with this approach is that this type of knowledge is often difficult to obtain for domains with many "subdomains" [7]. It is hard to know how many of these subdomains are necessary. In addition, the system has to be able to relate its visual input, which in this case is quite limited, with the correct subdomain. A second approach is to gather examples of failure and use inductive learning techniques to hypothesize which feature in the environment was responsible for the problem [18]. This may identify the feature, but it requires many failures and also gives no hint as to the appropriate action. A third approach is for the system to experiment with its available operators to see what actually works [4]. This approach can be quite effective, but it also can be quite time consuming and possibly dangerous. A fourth approach is to open up the robot and reprogram it. This requires skilled programmers and may be difficult if the robot is remotely located. A fifth approach involves increasing the interaction between the human and the robot so that the human can point out relevant features in the environment and associate them with the potential success or failure of a given operator or set of operators.

This latter approach builds on previous work in Soar on recovery from incorrect knowledge [12]. In Soar, recovery is complicated by the fact that chunking is the only learning mechanism, and it only adds knowledge to long-term memory, never modifying existing productions. In our original work, we demonstrated that it was possible to learn new productions that created preferences to correct decisions. The first step in this approach is to notice that an incorrect decision has been made. When Robo-Soar attempts the problem with the prism, the productions it learned on the first task leads it through the first four panels of Figure 5. It correctly moves the gripper above the block. At this point it then approaches the block, closes the gripper,

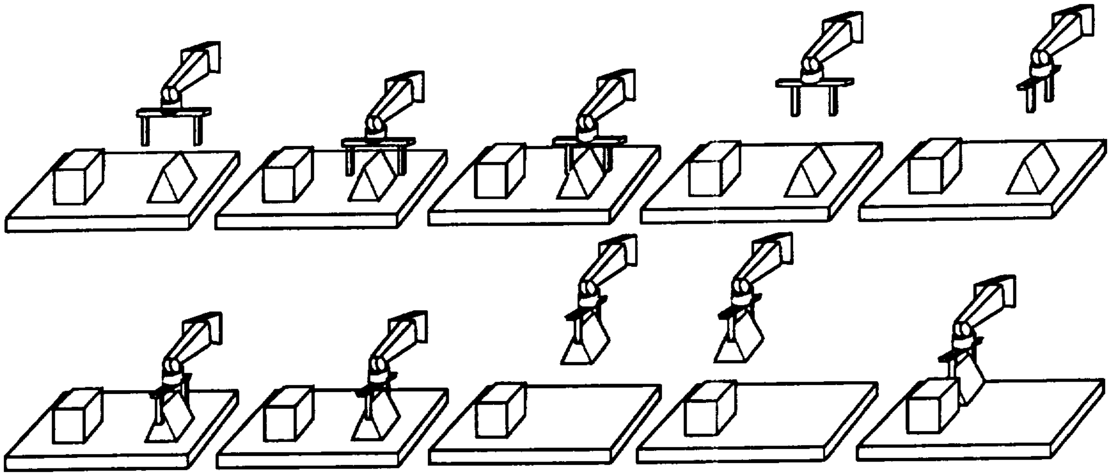

Figure 5: Trace of operator sequence using recovery.

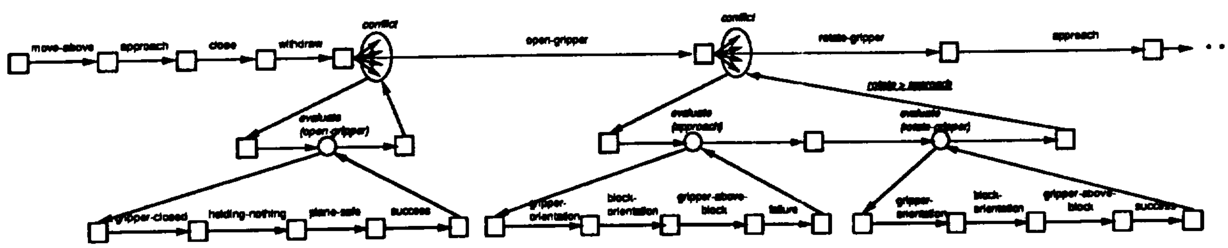

Figure 6: Trace of problem solving with recovery, omitting the advice problem space.

withdraws the gripper and the moves the gripper out of the way of the camera. After this last move it notices that the block is still on the table, so that at some point along the way, an incorrect decision was made.

If no additional knowledge is used, Robo-Soar encounters an impasse because it is in a new situation: the gripper is closed and holding nothing. If we advise it to open the gripper, it will immediately discover (incorrectly) that by approaching the block it will get on the path to the solution. Even though it knows there is an error, it will repeat its attempts to pick up the block indefinitely. This is because its internal knowledge is incorrect. None of its operators, such as approach or close, are sensitive to the orientation of the prism.

To avoid repeating the incorrect actions, domain-independent knowledge is added to Soar that attempts to recover from errors. The general strategy is to not trust the internal knowledge, forcing the system to reconsider each decision and accept guidance from outside. As before, the guidance includes suggestions of appropriate actions, but this time it may include suggestions of inappropriate actions as well as suggestions as to when an inappropriate action should be avoided. The remaining paragraphs of this section go through the details of this approach to recovery.

Reconsideration is implemented by forcing impasses for every decision. These impasses are forced by creating a dummy operator, ***deliberate-impasse***, and then creating preferences that make it both better and worse than the other operators. This is guaranteed to force an impasse and allows for the reconsideration of the decision in the resulting subgoal. In our example, the first action to take is to open the gripper. We skip over the process for doing that and go to the situation after the gripper is open as shown in Figure 6. A chunk from the previous problem creates a "best" preference for **approach**. However the conflict impasse prevents its selection and allows reconsideration.

Within the conflict subgoal, the selection space is chosen and internal operators are created to evaluate all

```
If the approach operator is applicable, and
   the gripper is above a block, and the gripper's orientation
   is different from a line in the middle of the block,
   and the rotate operator is available,
then create a preference that rotate is better than approach.
```

Figure 7: Example production learned by Robo-Soar.

of the external operators that were legal for the original state, the same as when an ordinary tie is encountered. At this point the human should not direct the system to the correct action. Even if that is found to lead to the goal, the knowledge to select **approach** still exists. Therefore, **approach** must also be evaluated so that it can be made "worse" than the correct operator, in this case **rotate-gripper**.

Once an evaluation operator for **approach** is selected, any directly computed evaluation is rejected because of a potential for error. Once a subgoal arises to compute the evaluation, a new problem space, called *examine-state* is selected. Its selection is predicated on the fact that an error has been encountered. The purpose of this problem space is to examine features of the state that are relevant to determining the appropriateness of the given operator for the current situation. If just "good" (or "bad") were indicated, the resulting chunks would be overgeneral and always prefer (or avoid) the operator, independent of context. To avoid this, the problem space consists of operators that examine and compare the features of the state, as well as operators that evaluate the state as being on the path to success or failure. Through interactions with a human, the appropriate features are selected and tested, and finally the operator is evaluated as useful or not. If it is deemed to be on the path to success, a "best" preference will be created for it and it will be selected to apply to the top state. However, as is the case for **approach**, failure is selected following the examination of the orientation of the gripper and the block it is above. In this case, **approach** will be avoided and the production in Figure 7 is learned that prevents its selection whenever the gripper is not aligned.

Following the evaluation of **approach**, **rotate-gripper** is evaluated, with the appropriate features of the state being tested before it is judged to be on the path to success. After **rotate-gripper** is selected, another conflict impasse arises, but in this case the human signals that the problem has been fixed and impasse is resolved by rejecting deliberate-impasse. From this point, the chunks apply and take Robo-Soar to the solution. When Robo-Soar encounters future problems, it immediately aligns the gripper before approaching a prism.

6 Discussion

The examine-state problem space is somewhat of a brute-force technique to learn new features. It requires an outside agent to lead the system through a search of potentially relevant features. Although it may not be considered the most elegant or complex machine learning technique, it allows the human to easily correct the system. In addition, this same approach can be used without an outside agent by having the system engage in experimentation. To experiment, the system can guess at relevant features. It will often learn to pay attention to irrelevant features, and thus create overgeneral chunks. But after many interactions with its environment, it will learn to ignore irrelevant features. If this search was a completely blind one, it would be similar to the search through hypothesis space performed by empirical learning techniques. One way to view the examine-state problem space is as an empirical learning method implemented on top of a deductive learning system.

Of course, the search for relevant features does not have to be blind. Many powerful heuristics are available, such as concentrating on new, unknown features, as well as those features that are modified by the operators under consideration. For example, if the system has discovered that the rotate operator is necessary, it could concentrate its search for features relevant to avoiding approach to those modified by rotate. Carbonell and Gill [4] have proposed more deliberate experimentation techniques that would be applicable for this task. Rajamoney and DeJong [19] have described a more elaborate approach to experimentation that is used to learn theories of physical devices.

The goal of our research was to demonstrate the practicality of learning using tele-robotics in a real domain.

Our actual task was quite simple, but Robo-Soar's ability to learn to solve these problems demonstrated the idea. Our current goal is to extend Robo-Soar to more complex tasks, expanding the spectrum of human interaction. At one end, we plan to investigate increasing the modes of communication, so that the user can interrupt Soar at any time to give advice, dynamically reprogramming the system through instructions. One of the original inspirations of the Soar project has always been to create an Instructable Production System where the system is never programmed, only given high-level advice [21,22]. We will also expand the current interface, so that it is more natural for the human to pick actions and relevant features. On the other end of the spectrum, we plan to study experimentation techniques so that Robo-Soar will be able to learn much of the same information on its own, when human advice is unavailable.

7 Acknowledgments

We would like to thank Karen McMahon for implementing Soar 5.0, and Mark Wiesmeyer for developing and implementing the Soar input and output interfaces. Without these extension to Soar, Robo-Soar would not have been possible. We also like to thank the staff of the Robotics Laboratory for help with the tele-autonomous software and hardware. Finally, we thank Paul Rosenbloom and Allen Newell for ideas relating to this work.

References

[1] P. E. Agree and D. Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of AAAI-87*, 1987.

[2] G. A. Boy and M. Delail. Knowledge acquisition by specialization/structuration: a space telemanipulation application. In *Workshop on Integration of Knowledge Acquisition and Performance Systems*, August 1988.

[3] R. A. Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, MIT, June 1987.

[4] J. C. Carbonell and Y. Gil. Learning by experimentation. In Pat Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 256–266, 1987.

[5] L. Conway, R. Volz, and M. Walker. Tele-autonomous systems: methods and architectures for intermingling autonomous and telerobotic technology. In *Proceedings of the IEEE International Conference on Robotics and Automation*, February 1987.

[6] G. DeJong and R. Mooney. Explanation-based learning: an alternative view. *Machine Learning*, 1(2):145–176, 1986.

[7] R. Doyle. Constructing and refining causal explanations from an inconsistent domain theory. In *Proceedings of AAAI-86*, Morgan Kaufmann, 1986.

[8] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[9] A. Golding, P. S. Rosenbloom, and J. E. Laird. Learning general search control from outside guidance. In *Proceedings of IJCAI-87*, Milano, Italy, August 1987.

[10] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell. Parallel implementation of OPS5 on the encore multiprocessor: results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[11] W. Harvey, D. Kalp, M. Tambe, A. Acharya, D. McKeown, and A. Newell. Measuring the effectiveness of task-level parallelism for high-level vision. Computer Science Department, Carnegie Mellon University, In preparation, December, 1988.

[12] J. E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of the National Conference on Artificial Intelligence*, August 1988.

[13] J. E. Laird and K. A. McMahon. Destructive state modification in soar, draft iv. 1989. Artificial Intelligence Laboratory, The University of Michigan, unpublished.

[14] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.

[15] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[16] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1, 1986.

[17] T. M. Mitchell, S. Mahadevan, and L. I. Steinberg. LEAP: a learning apprentice for VLSI design. In *Proceedings of IJCAI-85*, pages 616–623, Los Angeles, CA, August 1985.

[18] M. Pazzani, M. Dyer, and M. Flowers. The roel of prior causal theories in generalization. In *Proceedings of AAAI-86*, American Association for Artificial Intelligence, Morgan Kaufmann, Philadelphia, PA, 1986.

[19] S. Rajamoney and G. DeJong. Active explanation reduction: an approach to the multiple explanations problem. In J. Laird, editor, *Proceedings of Fifth International Conference on Machine Learning*, pages 242–255, Morgan Kaufmann, Ann Arbor, MI, 1988.

[20] S. Rajamoney and G. DeJong. The classification, detection and handling of imperfect theory problems. In *Proceedings of IJCAI-87*, pages 205–207, Morgan Kaufmann, Milano, Italy, 1987.

[21] M. D. Rychener. Approaches to knowledge acquisition: the instructable production system project. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 228–230, AAAI, August 1980.

[22] M. D. Rychener. The instructable production system: a retrospective analysis. In *Machine Learning: An artificial intelligence approach*, Tioga, Palo Alto, CA, 1983.

[23] A. M. Segre. *Explanation-Based Learning of Generalized Robot Assembly Plans*. PhD thesis, University of Illinois at Urbana-Champaign, 1987.

[24] A. M. Segre. Explanation-based manipulator learning. In *Proceedings of the Third International Machine Learning Workshop*, pages 183–185, Skytop, PA, 1985.

[25] L. Suchman. *Plans and Situated Action*. Cambridge University Press, 1987.

[26] M. Tambe, D. Kalp, A. Gupta, C.L. Forgy, B.G. Milnes, and A. Newell. Soar/PSM-E: investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, July 1988.

[27] M. Wiesmeyer. Soar I/O reference manual, version 2. 1988. Artificial Intelligence Laboratory, The University of Michigan, unpublished.